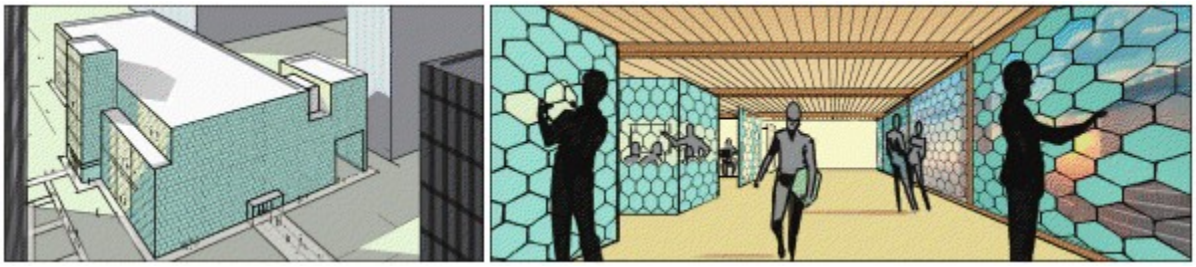
Spring 5-1-2011

Realistic simulation of spatial computers and robot swarms

Karamjeet Singh Khalsa
University of Colorado Boulder

Realistic Simulation of Spatial Computers and Robot Swarms

Karamjeet Khalsa
Department of Computer Science
University of Colorado, Boulder



Working with Charles Dietrich, Department of Computer Science

Fall 2010 – Spring 2011

Faculty Advisor:
Nikolaus Correll, PhD
Department of Computer Science
University of Colorado, Boulder

Thesis Committee:
Michael Main, PhD, Department of Computer Science
Rick Han, PhD, Department of Computer Science

A thesis submitted in partial satisfaction of the
requirements for the degree Bachelors of Science in
Computer Science

1. Abstract

The goal of Amorphous Computing is defined as: “To identify organizational principles and create programming technologies for obtaining intentional, pre-specified behavior from the cooperation of myriad unreliable parts that are arranged in unknown, irregular, and time-varying ways” [1]. Amorphous Facades are stationary formations of amorphous computers used in building environments and are constructed as a wall. One of the desired functionalities of the Amorphous walls is to be able to track occupancy within an interior environment.

Pymorphous is a spatial computing library for Python. Currently, Pymorphous has its own simulator, but the simulator is very abstract and doesn't realistically simulate physical robots or device hardware limitations. Webots is a virtual robot simulation program that is much less abstract that the Pymorphous simulator and that accurately simulates physics and realistic hardware. The simulator-runtime for Pymorphous is very specific to its own simulator. To allow Pymorphous to be simulated in a less abstract environment, Webots, I will create a new runtime which will facilitate communication between amorphous computing robots within Webots and Pymorphous.

To demonstrate the functionality of the Webots-runtime for Pymorphous, I will develop three simulations within Webots. A simple neighborhood simulation will be used to show the functionality of Pymorphous neighborhood calculation between amorphous wall panels in Webots. A velocity tracking simulation will be used to demonstrate the functionality of simple tracking algorithms within Webots, similar to algorithms that the wall might actually use to track occupancy. Lastly, the setup of the Amorphous Wall within Webots will be changed to reflect mobile robots to illustrate the ability of Webots to simulate more complex Pymorphous flocking algorithms on mobile robots.

2. Introduction

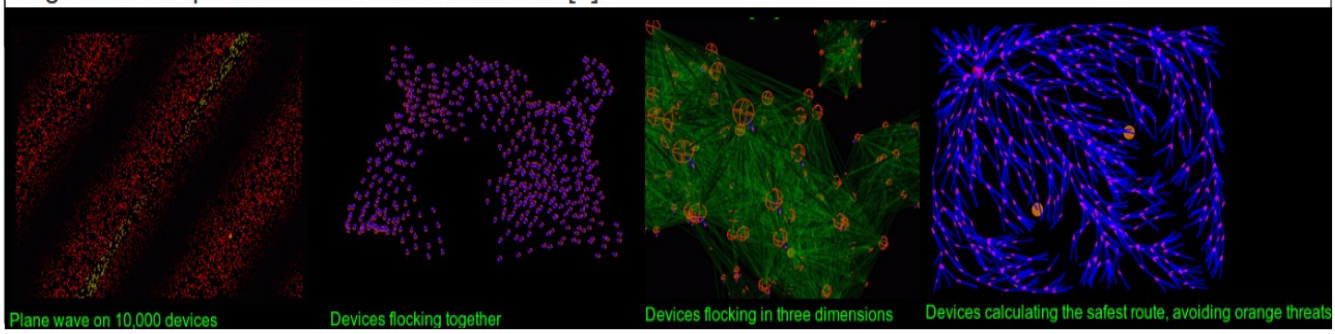
2.1 Amorphous Facades

Self-organizing amorphous facades can be used to optimally control various parameters that affect an interior environment. Lighting (and consequently thermal energy) can be varied by adjusting transparency. Permeability of the facades can be varied to minimize energy use with respect to HVAC systems. Individual units that make up the facades agree upon an optimal parameter configuration through a combination of user input, environmental sensors, local communication, and other distributed algorithms. Amorphous Facades will use occupancy tracking and gesture recognition to make behavioral decisions. Currently, there are some basic hardware demos of the walls, but they are buggy and their small size makes them inaccurate in terms of modeling high fidelity amorphous walls in actual environments. Accurately simulating the demo hardware within the amorphous wall in Webots will provide clean, large scale, and high fidelity visualization, while allowing the hardware to be debugged in a virtual setting.

2.2 Flo/Proto

Proto (or Flo, a newer alternate implementation) is the driving force that lies behind the distributed algorithms of the Amorphous Facades. Both Proto and Flo are based on a structural framework designed specifically to model group behavior in continuous space [3].

Figure 1: Snapshots of MIT-Proto in Action [3]



Although Proto's continuous space model might not be useful in the sense of the wall robots' own positions, as they are stationary, it could be helpful in terms of monitoring the continuous space of other objects in the room. For example, one of the functionalities of the wall is occupant tracking, which would be used in order maximize efficiency. Specific and precise occupant information would allow the wall to make various decisions, such as dimming the lights in an empty room or turning the heat down in rooms that are generally unused.

It is well known that this sort of occupant tracking is no easy task, especially as the number of occupants being tracked increases [4]. Proto's distributed algorithms propose a way of, grouping, reconciling, and making decisions based on a vast amount of data from multiple sources, which can help maintain accurate tracking. Additionally, monitoring the room in terms of continuous vector position fields allows a more expansive set of positional probabilities to be evaluated and results in a more comprehensive view of the state of the environment.

2.3 Pymorphous

Pymorphous is a spatial computing python library written by Charles Dietrich and inspired by Proto. Pymorphous aims to deliver the same functionalities of Proto, that is: the group-behavior algorithms and continuous space model, but through the imperative language of Python [6]. The quantitative performance differences between Proto and Pymorphous are still not fully tested, but Pymorphous has demonstrated its ability to mirror Proto's functionality in a variety of areas. These include: neighborhood functions, such as gradient and int-hood, as well as more complex group behavior algorithms such as flocking and line detection.

Additional differences between Proto and Pymorphous include: the increased readability and write-ability of Pymorphous, as well as the ability of Pymorphous to use recursion and dynamic message sizes. Functional languages, such as Proto, certainly have their own advantages over imperative languages- such as being able to accomplish complex tasks using very succinct statements. Unfortunately, these advantages are seldom realized, as only highly specialized groups are typically able to even understand what the statements are doing, due to the unique way that functional languages

treat data. This effect is amplified by Proto's own unique syntax, creating an environment that is all but impossible for even an average educated person to understand, let alone use to accomplish a goal. Through the medium of the well known imperative language of Python, Pymorphous delivers an experience that is much more user friendly than Proto, while still providing its powerful group-behavior algorithms and continuous space model.

2.4 Webots-- Hardware Simulation Environment

Webots is a virtual simulation environment that provides capabilities for modeling and visualizing robots (along with their many parts, such as LEDs, sensors, cameras, wheels), and then observing their behavior in detailed environments, possibly along with other robots. The software and hardware of the robots can be easily modified and manipulated to a high degree of precision. The malleability of: the physical and electronic specifications of the robots, their software, and the environment in which they act, makes Webots an ideal simulation environment for the amorphous wall. Additionally, the built in representations of physical dimensions, physics, and electronic limitations yield a simulation environment much less abstract than that of the default Pymorphous simulator.

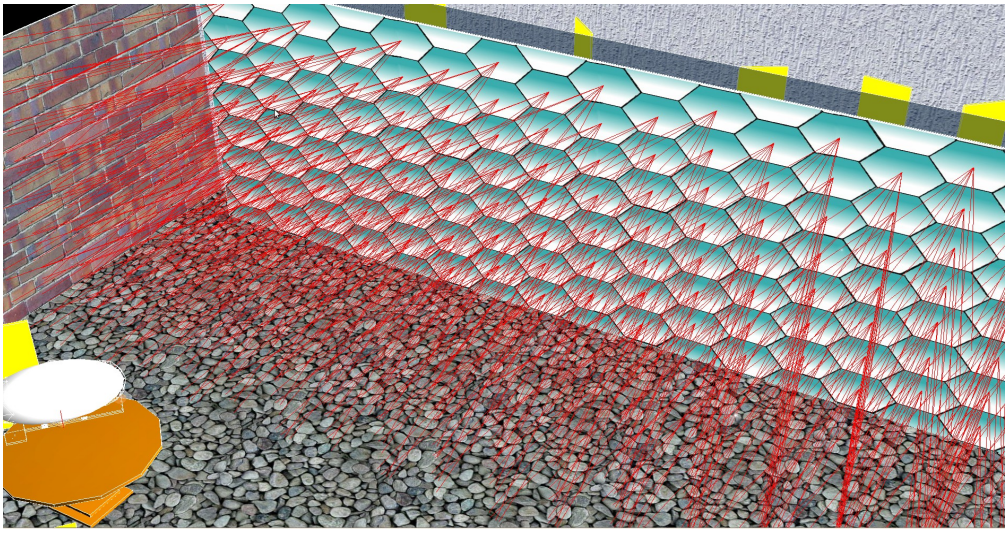


Figure 2: Webots world implementing the amorphous wall as a hex grid with sensor rays shown

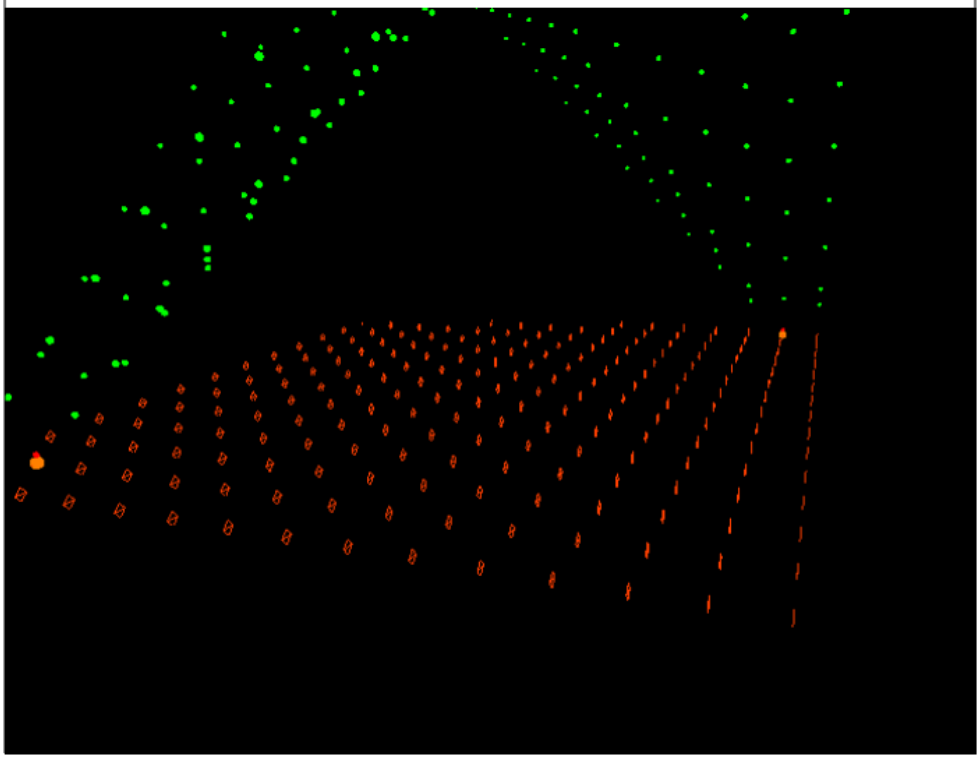
3. Materials and Methods

3.1 Pymorphous-- Simulator and Runtime

Like Proto, Pymorphous has its own simulator that can be used to visualize the effectiveness of its algorithms. The simulator is similar to Proto's, both visually and functionally. Both simulators are very abstract, that is, they each do very little to account for realistic environmental rules and laws. Neither simulator has any concept of the electronic implementation of the individual devices, the hardware mechanisms behind sensing and inter-device communication, or the physical representation of both the devices and the environment. Physical constants such as gravity, friction, inertia, and object collisions are also ignored.

The Pymorphous simulator communicates to the Pymorphous library through a simulator-runtime. The simulator-runtime encapsulates the functionalities that are reliant on aspects of the simulation environment and delivers information about the environment to the Pymorphous library for use in the group-behavior algorithms. These functionalities include: communication between the devices, calculation of the neighborhoods of which the devices belong to, controlling the sensing functionality of the devices, monitoring the velocity of each device and updating its location accordingly, as well as maintaining local state of each device. The state of each individual device is typically communicated to the user via LED objects that can change the color of the device in the simulator.

Figure 3: Gradient Demo in Pymorphous Simulator



3.2 Webots World Layout

Webots worlds consist of objects, their properties, their child objects, the properties of their children, and so forth. For example, a robot node in the wall simulation has a base physical location, a software controller that controls its actions when the simulation is run, as well as any number of child hardware devices. In this example, the hardware devices making up a wall node might be a distance sensor (with its own properties), and emitter and receiver nodes to communicate with its neighbors. The set up of the world is stored very intuitively in a text file, with brackets and indentation representing child/parent hierarchical relationships.

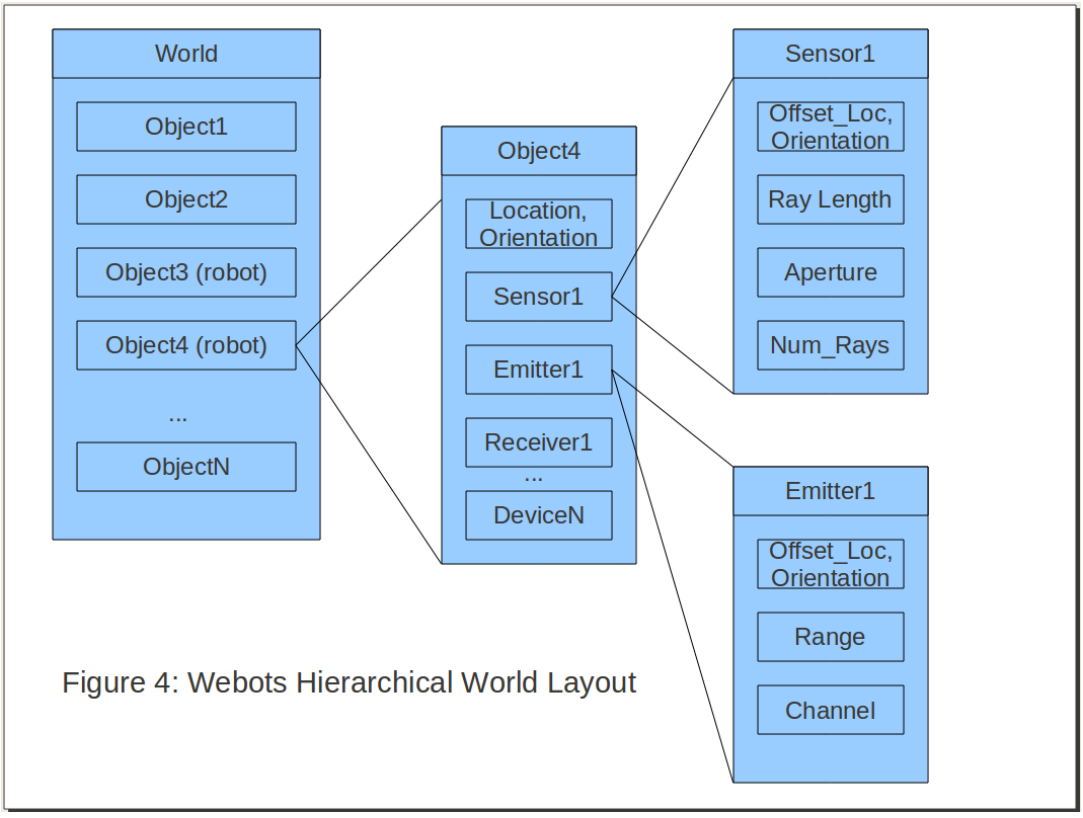


Figure 4: Webots Hierarchical World Layout

3.3 Webots Text Generator

The nature of amorphous computing lends itself very well to the text-based set up of the Webots simulation environment. Namely, the requirement that all of the computational particles/devices share the same hardware and software specifications. Rather than needing to individually modify/update each of the many devices manually when a change in the devices hardware is desired, regular expressions can be used to find and replace/change specific elements of the world automatically. This was the motivation behind the Webots Text Generator.

The Text Generator takes the text based layout of a Webots sample robot as input. This sample robot is meant to represent the updated hardware layout of a robot that is to be distributed into the Webots world. Next, the generator repeatedly appends copies of the robot into an output file. Properties of the individual output robots that are desired to be unique, such as location or orientation, are modified on the fly via Python regular expressions. Under other conditions, such as with real robots, updating the hardware of a large group of robots could be very time consuming. This automatic distribution makes Webots even more attractive as a testing environment and can save time and money when compared to iterations of buying, distributing, and testing hardware changes in a group of real robots.

3.4 Webots Controllers

The software that determines robot behavior once a simulation is run lies in a source file known as the “controller”. Webots is compatible with a wide variety of languages for the controller including C++, C, Java, Matlab, and Python. The name of a controller is a property of the robot and is essentially a pointer to the source file. By definition, all of the amorphous computing devices share the same software code. This attribute of Amorphous Computing in conjunction with the Webots controller-pointer layout allows changes to the controller to be distributed to all of the devices instantly. Similarly to the hardware setup, Webots software distribution is fast and efficient and can save countless hours of software testing iterations when compared to actual robots.

The specific layout of a controller itself consists of a controller class which contains an initialize() method and a run() method. The initialize method is called at the start of the simulation and links the various devices that make up the robot to data structures, so that they can be manipulated as necessary through functions in the code. The run method contains an infinite while loop that performs the algorithms that the robot is to run during each time step. At the end of the while loop in the run method, there is a call to step(), which lets the Webots world know that the controller has completed its operations for the given time step. The step method also breaks the while loop when the simulation is quit.

3.5 Webots-runtime

The Webots-runtime provides a layer of abstraction in between Webots robot operations and Pymorphous library functions. The runtime encapsulates neighbor communication, LED controls, move method, sensor information, a step function, and any other Webots related device functions. The main function of the Webots-runtime is to allow Pymorphous to interact with Webots on the level of its own simulator, without needing to know that it's actually a different simulation environment.

Figure 5: Standard Webots/Controller Relationship

1. Webots calls each controller's code when the simulation starts.

2. The controller creates an instance of the robot object and maintains links to the robot's devices so that it can interact with them.

3. Webots contains a reference to the Robot object.

4. During each time step, Webots iterates through the robot objects that have been created and calls their Run() methods.

5. The Robot object's run method calls the controller's run method and allows it to execute each time step.

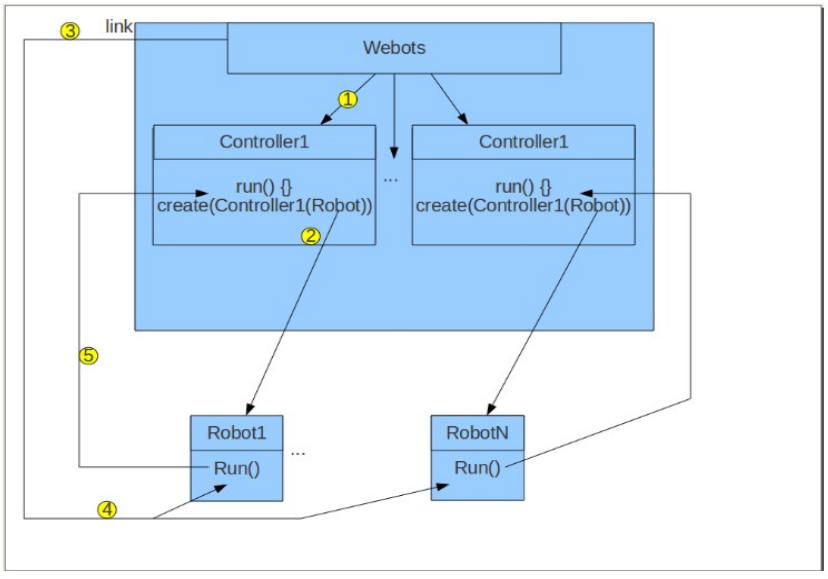


Figure 6: Webots/Controller Relationship with Pymorphous and Webots Runtime

1. Webots calls each controller's code when the simulation starts.

2. Each controller calls the spawn_cloud() method in the Webots runtime, passing an instance of itself as well as its robot object in Webots.

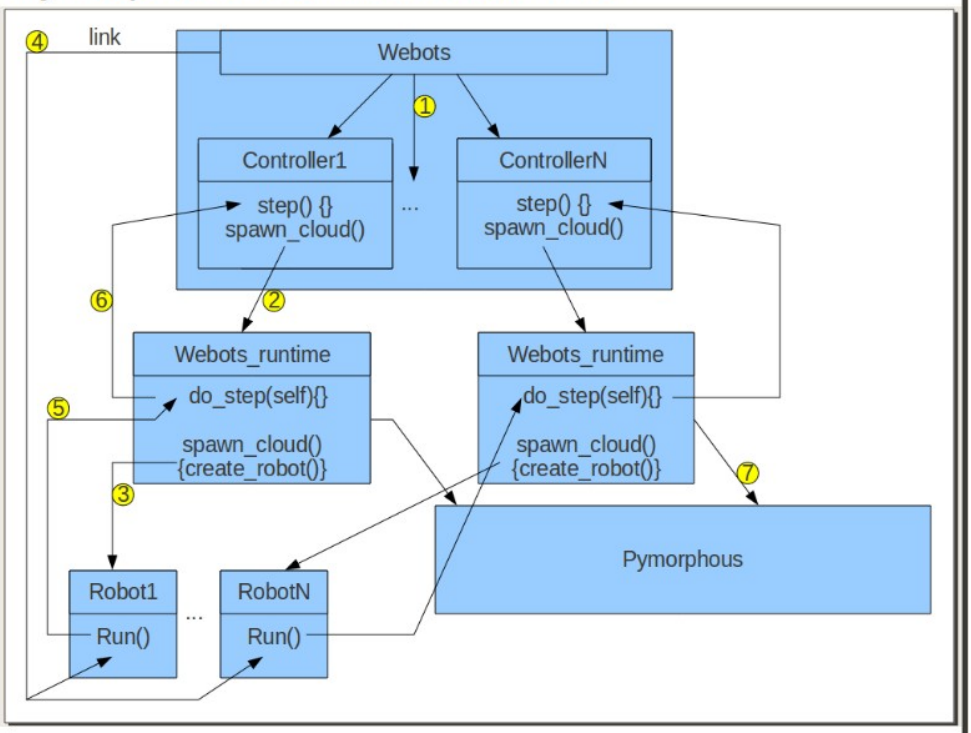
3. The Webots_runtime creates an instance of the robot and maintains the link to the robot so that it can interact with it.

4. Webots retains a link to the robot object and calls each robot's Run() method during each time step.

5. Each robot's Run() method calls the do_step() method in the Webots_runtime.

6. The Webots_runtime calls the controller's step method each time step.

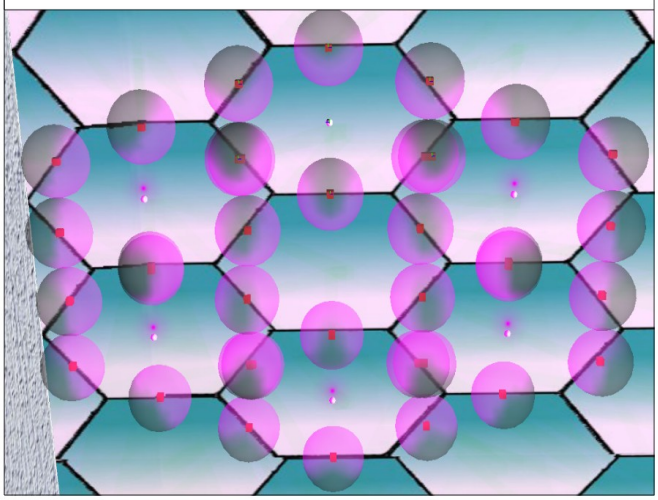
7. The Webots_runtime can reference Pymorphous functions as needed.



3.6 Communication in Webots and in the Webots-runtime

Communication in Webots between robot devices in the Amorphous Wall occurs via emitter and receiver devices. Each emitter/receiver pair is meant to approximate a physical serial connection between adjacent nodes. In Webots, the emitter and receiver nodes are actually infra-red communication devices, but for all intents and purposes can be assumed to be direct serial connections, as the wall elements are stationary and always within direct communication range of each other. Additionally, each emitter and receiver node can be fully customized to reflect the physical properties of the actual communication device used in the wall.

Figure 7: Communication Layout of the Wall in Webots



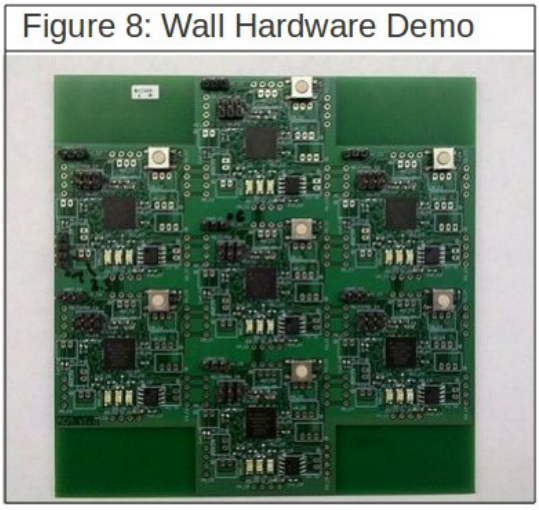
Communication is fully encapsulated within the Webots Pymorphous runtime. As the Pymorphous simulator has different methods of communication, Webots emit and receive calls lie within the shared

functions init(), dostep(), and nbr(). The emitters and receivers are set up and activated within the init method. Each time step, the Webots-runtime serializes any messages that a device wishes to emit and sends them to Webots. Webots fills the buffers of receivers with the messages they have been sent and then Webots-runtime pulls the messages out of the buffers and unpacks their contents.

Serialization of messages occurs via the Python pickle module. The use of the pickle module was intended to allow debugging of the wall hardware's serial communication through Webots. The setup of the emitters and receivers as data streams with buffers should support debugging efforts, as they are very straightforward and permit detailed message observation. Additionally, the buffer sizes and baud rate are fully customizable.

3.7 Wall Hardware

Figure 8: Wall Hardware Demo



The current demo model is simple, but bugs can be plainly observed. Currently, simple sequences of bits are passed locally in between the 6 self contained edge units. These messages are treated as instructions, which at this stage simply blink the unit's LEDs (either red or green). Additionally, when instructions are acted upon, they are forwarded/passed along to adjacent nodes. Thus, it is trivial to judge the functionality of the communication, as a single green blink should traverse around the set of units. Currently, certain nodes do not pick up the communication, and thus do not blink their LEDs accordingly. It is postulated that this is due to corruption over the serial connection. The proposed solution is to ensure that the serial communication protocol is implemented in such a way that the possibility of corruption is minimized and, when corruption is found, to handle it. Webots supports very fine tuning of the device communication. Additionally, the serialized messages create a friendly virtual environment where debugging and corruption management efforts may begin.

3.8 Webots-- representation of the wall

At first, the plan was to set up the wall in Webots as a single robot. The robot would have multiple sensors (each representing a singular unit). The code controlling the entire wall unit would be responsible for reading in all of the sensor values, as well as handling communication in between the local sensor regions, and eventually coming up with neighborhood readings (through Pymorphous), broadcasting the readings to other local neighborhoods, updating again, and so forth. It was quickly decided that this single robot would be an inaccurate representation of the amorphous medium, and would ignore the primary strengths of Pymorphous. These include: modularity, simplicity of each unit, and the computational power of the group of individuals working together (much greater than a sum of its parts). Additionally, it would require immense effort to translate the demo's controller software to such a mismatched representation. Breaking each unit of the wall into its own robot yields a much more accurate depiction of the hardware demo.

3.9 Webots-runtime NBR Function

The neighbor function in the Webots-runtime references a list of neighbors, with fields associated with each of the neighbor values. The actual data structure used to store the neighbors and fields is a Python dictionary, with the keys of the dictionary being unique neighbor ids and the values being the fields of the neighbor. The fields associated with each neighbor can consist of any number of variables, which are typically continuous. Some example field variables could be: the coordinates of the node, timestamps, or various vectors used algorithmically.

Each time step, the Webots-runtime parses any incoming messages received within Webots into field, neighbor-ID pairs. Next, the neighbors dictionary is updated. New neighbors are added to the neighbors dictionary, along with their respective fields. For neighbors that already exist in the neighbors dictionary, the field values are replaced with the new, more recent, fields. Additionally, each neighbor is timestamped with the last time it was updated. When a neighbor has not been updated over a period specified by a global timeout variable within the runtime, it is purged from the neighbors list.

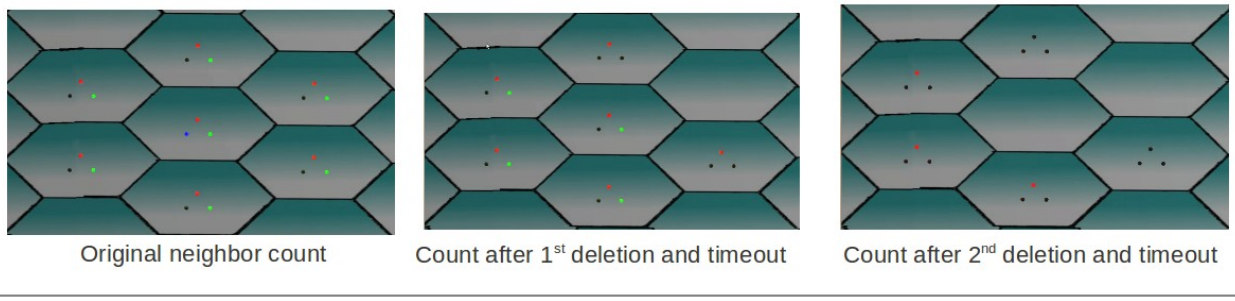


Figure 9: Neighbor Demonstration- The blue LED indicates 6 or more neighbors, the green LED indicates 3 or more and the red LED indicates 2 or more. As neighbors are deleted and the timeout reached, neighbors are purged and the neighbor count is decreased.

3.10 Wireless communication

For the flocking simulation, in which the wall devices are designed using mobile robots, a device has to be able to deal with an arbitrary number of neighbors. Mobile robots, instead of using direct serial connections, use a single wireless device that can communicate with all of the other relevant devices. Thus, the six emitters and receivers must be replaced by a single emitter and receiver, representing a wireless device. Additionally, all of the communication devices operate on the same channel, as all of the robots are meant to be able to communicate. The type of the emitters and receivers was chosen as infra-red, but Webots also supports radio devices, which could be chosen instead if they provide a more accurate representation of the physical hardware.

To test the wireless communication devices, they were first tested on stationary devices in the wall simulation in Webots. Strangely enough, when using the wireless communication method, the devices were only picking up on a subset of their neighbors. Additionally, the number of neighbors detected was inversely proportional to the number of fields being sent to the neighbors.

After additional inquiry, it was discovered that the buffer size of the wireless receiver was too small to hold the communication data of all of the neighbors. Once it was full, the communication of any subsequent neighbors was simply discarded. The direct serial communication did not have this problem because each of the six receivers had its own communication buffer, so the total communication buffer size was six times that of the wireless. In terms of the fields, each field being sent takes up space on the

receiver's buffer. Thus, the more fields that are sent, the larger the buffer needs to be to hold them. This is perfectly realistic.

The data buffer size was increased from 4096 bytes to 8192 bytes and all of the stationary neighbors were detected, even in the case of three fields being communicated each timestep. The maximum number of neighbors tested to be working correctly in this simulation was six. It was observed that this data buffer was fairly large, so additional testing may be required to determine if there is additional overhead contributed by Webots in terms of the size of the data. Overall, however, this property of the receiver device ended up being a good illustration of the ability of Webots to simulate realistic physical limitations.

3.11 GPS and Compass

To calculate a device's coordinates within Webots, the device uses a GPS. In later simulations, a compass is used to calculate orientation. One of the requirements of amorphous computers is that they have no direct information about their environment; only local, relative, observations are used to evaluate performance. Upon first glance, a GPS appears to be a clear violation of this requirement. The GPS devices, however, are only used to calculate relative positioning in between the devices, not absolute position. In the same way, compasses are only used to calculate relative orientation. Thus, a GPS is simply being used to approximate actual relative positioning devices that might be used by actual amorphous robots.

3.12 Webots-runtime Move Command

The move command in the Pymorphous simulator-runtime is very simple and doesn't have any notion of the actual physical mechanism of movement. Each timestep, the simulator-runtime simply increments the locations of the devices with their velocity and then reflects their new positions to the user. This is a very abstract simulation.

To define the move command to function realistically within Webots using a differential wheels robot, several additional things need to happen. First, a robot needs to have an idea of its orientation; to do this, Webots supplies a compass. As mentioned earlier, the compass is not a violation of amorphous computing principles, as it is only used to calculate local, relative orientation. Next, the robot needs to turn to face the desired orientation; actual wheel speeds must be specified in order to turn. Finally, once the robot is facing the correct direction, the wheels can be set at a “cruising speed”, with or without a gradual turn, and can maintain that speed until a new direction is required. Finally, since robots can easily run into each other and get stuck, an algorithm for object avoidance must be defined.

As mentioned before, the Webots-runtime maintains a link with the robot in Webots. This allows the Webots-runtime to control the wheel speeds of the robot via the differential wheels setSpeed() method. The setSpeed method takes two floats as arguments, which respectively represent desired left wheel and right wheel rotational speeds.

The move method works by first checking if the robot has been stuck for a constant number of timesteps. Stuck is defined as non moving, which could be monitored in a real situation via wheel encoders reporting wheel turn. When it is discovered that a robot is stuck, the robot then backs up a random amount while turning. The direction of the turn is based on the difference between the sensor readings of two distance sensors placed on the front of the robot, used to detect physical objects in the robot's path. For example, if the distance sensor on the left is greater than the distance sensor on the right, then the obstacle is located slightly more to the left of the robot so the robot will turn to the right

as it is backing up. After the backup turning period, the robot will go forward a small random amount of timesteps to attempt to distance itself from the collision area.

When a robot is not stuck, then it actually uses the velocity passed into the move function. The radian value of the robot's current orientation is calculated using the arctan function atan2. atan2 is a similar to the arctan function, but is modified in some ways to make it more suitable for applications involving vectors in Euclidean space [2]. Next, the robot's desired radian orientation is calculated using the atan2 function on the velocity argument. Next, the radian difference between the current orientation and desired orientation is calculated. If the difference is greater than a constant, epsilon- specified in the runtime, then the robot turns, setting the wheel speeds to (10*delta, -10*delta), where delta is equal to the radian difference. As the turn speed is based on the radian difference, the robot automatically slows down the turn as it is approaching the correct orientation. Once facing the correct orientation, the robot sets both wheels to a constant cruising speed based on the dot product of the input velocity. The robot continues in that direction until either the desired velocity vector changes or it gets stuck.

4. Experiments

4.1 Tracking Object Velocity Using the Wall

Now that the foundation of neighbor communication through Webots has been established, the next step is to test relative object coordinates through a more involved example. Additionally, as one of the desired functionalities of the amorphous wall is occupant tracking, some sort of illustration demonstrating the ability of the wall to track an object using vector calculations would certainly be a step in this direction. The product was the tracking-velocity simulation.

The goal of the tracking-velocity simulation was to be able to approximate the velocity of a moving object in front of the wall using Pymorphous group-behavior algorithms. The moving object chosen was a blimp that flies based on prespecified physics within Webots. Wall devices detect the blimp using distance sensors. Although, in actuality, the distance sensors in Webots returns an integer related to the distance of the blimp from the device (as actual distance sensors do), the algorithm simply uses the distance sensors as boolean variables. Either the blimp is detected, or it is not. This demonstrates the power of Pymorphous algorithms despite modest hardware and limited information.

4.2 Tracking Velocity Algorithm

The tracking-velocity algorithm works locally between neighbors. Once a device sees the blimp, it flags itself as “tracking” and stores the time that tracking first began. A tracking device will maintain tracking status for a period of time after it has lost sight of the blimp, specified by a timeout variable in the tracking_velocity controller. This timeout allows velocities to be postulated, even if the times at which the neighbors sense the blimp don't directly overlap.

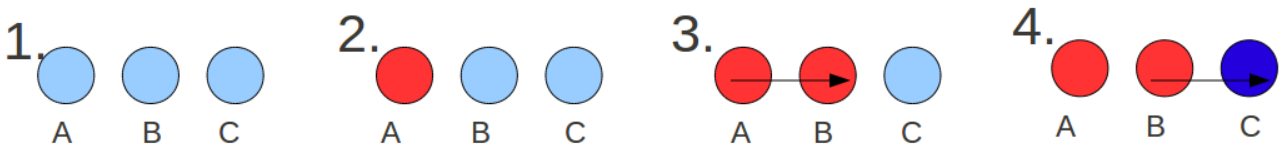
During each time step, each device sends a tuple representing fields to each of its neighbors. The fields include the device's coordinates, its tracking_start_time, and a delta vector (to be explained shortly). If the device is not tracking then tracking_start_time is equal to -1 and the delta vector is empty.

Additionally, during each time step, the device loops through each of its neighbors and pulls their respective tuple fields for use algorithmically. When a device is tracking, it looks through its neighbors for another device that is also tracking and that also has a tracking start time that is less than its own tracking start time. The device saves the coordinates of either the oldest or most recent (depending on the algorithm) neighbor that meets both of these requirements. From the saved neighbor's coordinates,

the device generates a delta vector equal to the difference between the neighbor's coordinates and the device's coordinates. The delta vector is then passed on to the device's own neighbors.

When a device sees that one of its neighbors has a delta vector that is nonzero, it checks whether the delta vector plus the neighbor's coordinates is close to its own position. When it is, the device marks itself as “is_next”, essentially expecting the blimp by predicting that its velocity will carry it to the devices location.

Figure 10:Tracking Velocity Algorithm



1. None of the devices detect the blimp.

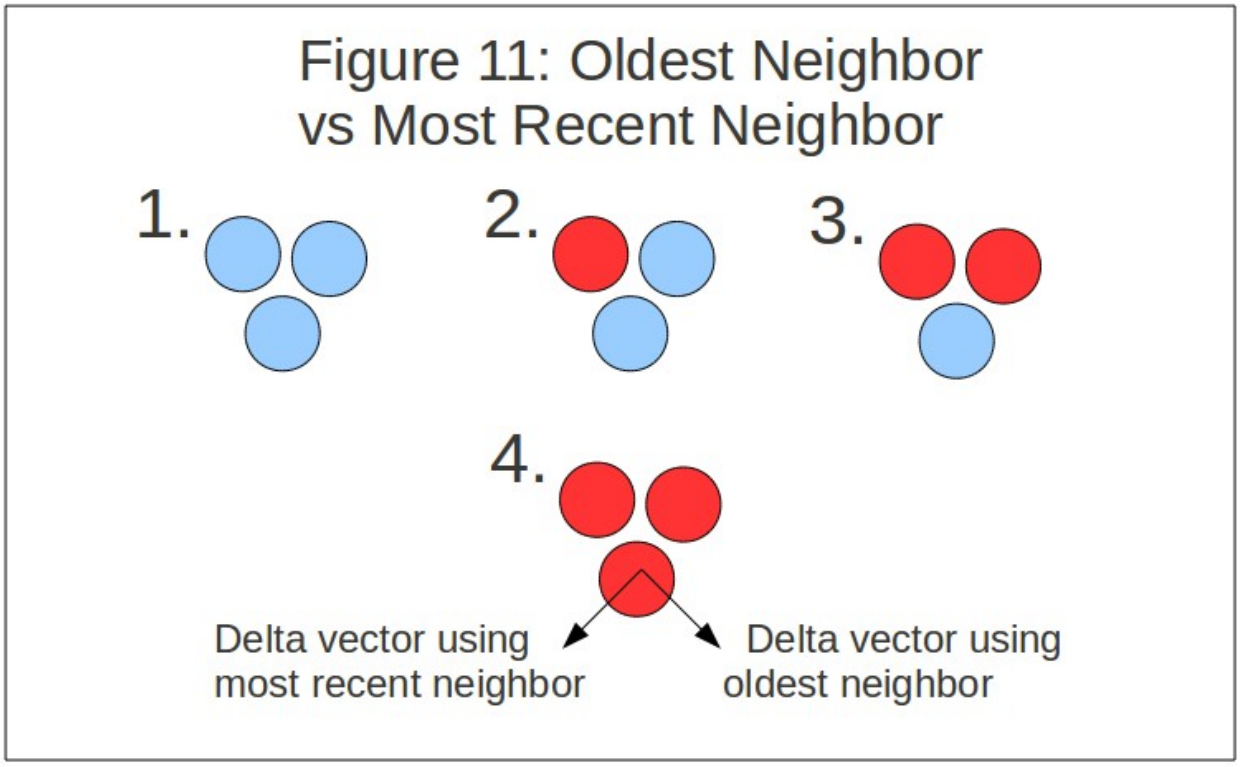
2. Device A detects the blimp, marks itself as tracking, and saves its tracking start time.

3. Device B detects the blimp, marks itself as tracking, and saves its tracking start time. Device B then sees that device A is not only also tracking, but has a tracking start time that is earlier than its own so device B then calculates a delta vector from A to itself.

4. The delta vector is sent to all of B's neighbors, who check to see if the delta vector + the location of B results in a location close to the neighbor's own coordinates. C sees that this is true for itself so it marks itself as “next”, predicting that the blimp's velocity will take it there.

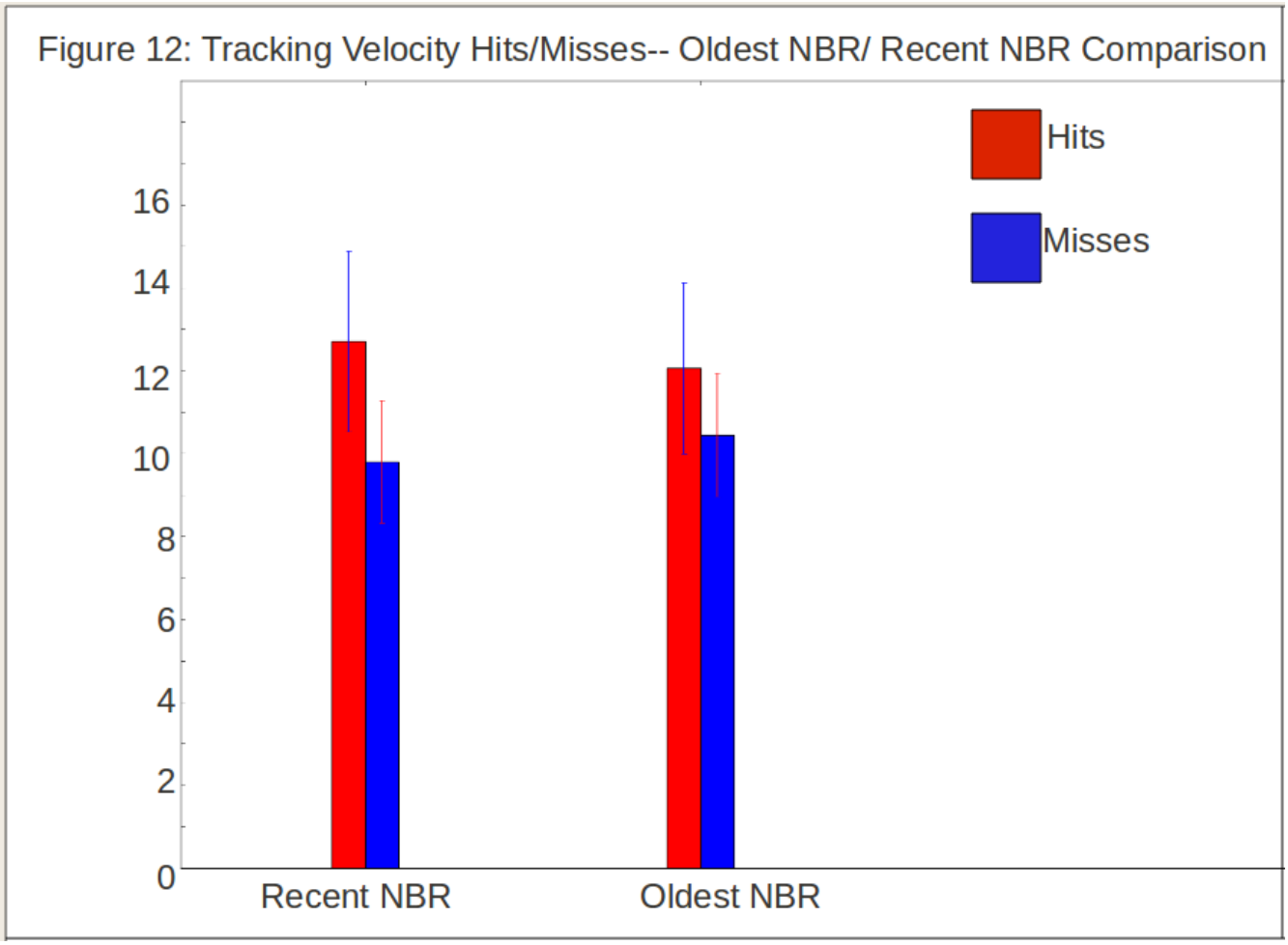
4.3 Oldest Neighbor vs. Recent Neighbor

In the case of more than one neighbor of a device having seen the blimp prior to the device itself, the device has a choice of which neighbor to use to generate the delta vector. A logical case can be made for choosing both the oldest neighbor or the most recent neighbor. Using the oldest neighbor might give a better “larger picture” view of the overall trajectory of the blimp by focusing on the greatest time difference between the device and its neighbor. It can also be argued that oldest neighbor yields outdated velocities and doesn't take advantage of more relevant data. Most recent neighbor assumes the blimp's velocity can be constantly changing and attempts to account for these instantaneous changes by calculating velocity using the most recent information available.



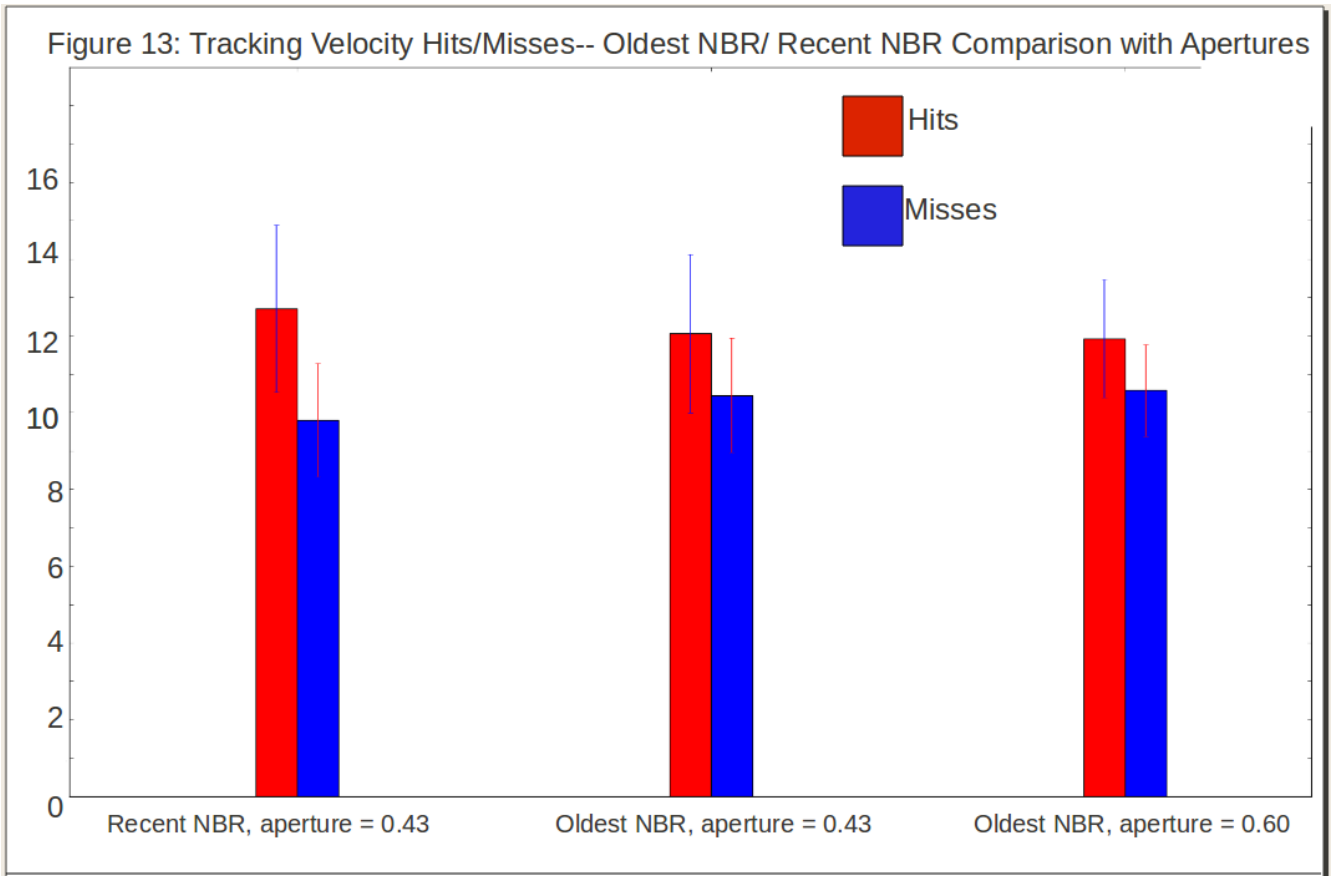
4.4 Performance of Oldest Neighbor vs Most Recent Neighbor

It is obvious that the delta vector generated using the oldest neighbor can differ greatly from the vector using the most recent neighbor. The performance of both the oldest neighbor algorithm and the most recent neighbor algorithm were measured over ten simulations using a hit-miss measurement. Essentially, if a device marked itself as “next” using a neighbor's delta vector, the device was marked as “expecting”, meaning that it expected that the blimp's velocity would bring it into direct sight. The device remains expecting until either it observes the blimp, in which case it counts as a hit, or it doesn't observe the blimp for a period of time (timeout) after the device is no longer being marked as “next”, which counts as a miss. The timeout period for these measurements was four seconds. During each simulation, the blimp was manually flown in front of the wall in a realistic random pattern. At the end of the simulation, the total number of hits and misses were calculated. The sum hits/misses were then averaged and normalized (as some of the simulations were longer than others), over the ten simulations.



4.5 Effect of Modifying Sensor Aperture on Tracking Velocity

To demonstrate the functionality of Webots to specify precise device properties and to show the effect on Pymorphous algorithms, the aperture of the sensors was modified and the tests were re-run. The aperture of the sensor is related to the angle between the sensor rays. Increasing the aperture widens the seeing-range of the sensor and allows the device to track the blimp when it is at an angle farther out of its field of vision.



4.6 Flock Tracking Simulation

To demonstrate the ability of the Webots-runtime to accurately encapsulate a move command that can be used by a differential wheels robot within Webots, a simple flock tracking simulation was designed. The flock tracking algorithm distributes known locations of the blimp as well as projected velocities of the blimp throughout the group of devices. When a device sees the blimp, it passes the information along to each of its neighbors in the form of fields, who use the information to generate a velocity vector and move toward the blimps location. The neighbors then pass along their own fields to each of their neighbors and so forth. The projected blimp velocity is taken into account when generating the velocity vectors, so the devices attempt to mirror the blimp's own velocity and flock around it. Additionally, timestamps are used to time out old information and weight newer information more strongly.

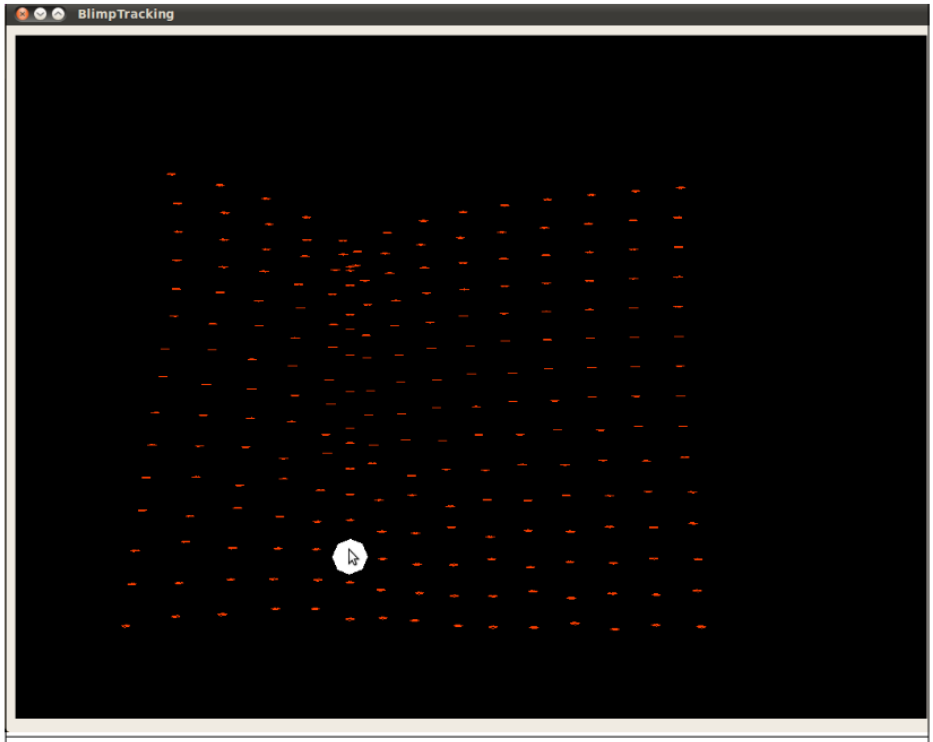
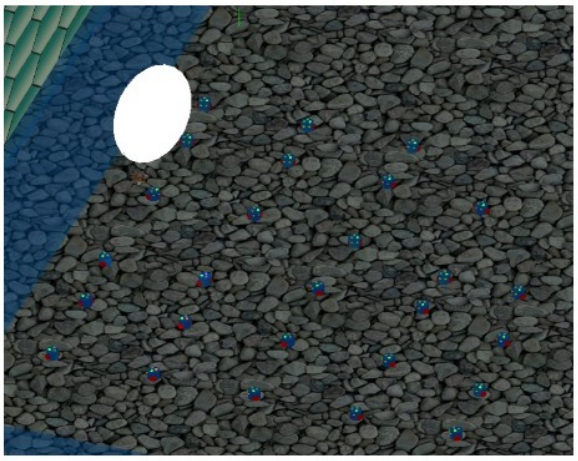


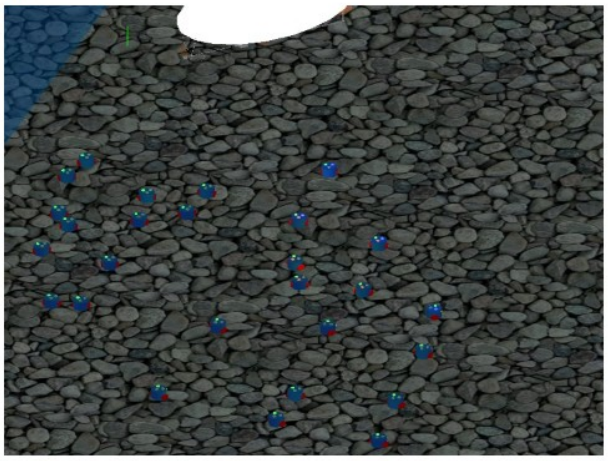
Figure 14: Flock Tracking Algorithm Within the Pymorphous Simulator

Figure 15: Devices Flocking in Webots

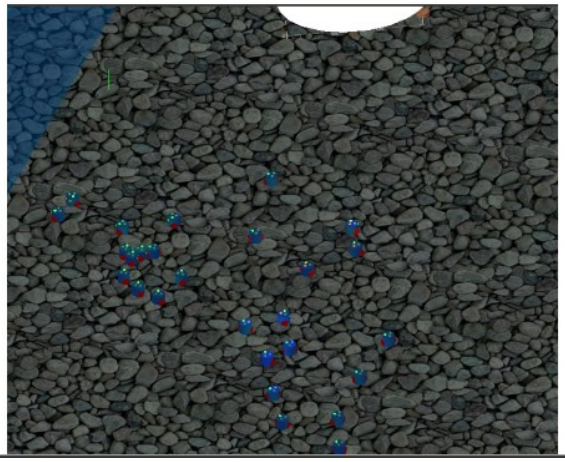
1.



2.



3.



5. Discussion

5.1 Oldest Neighbor vs Newest Neighbor

Originally, I was predicting oldest neighbor to perform better, as it assumes a more constant velocity, which I believed was more consistent with the behavior of the blimp. Surprisingly, the number of hits were higher using the Recent NBR algorithm and the number of misses were lower. I attribute this to the blimp's flight pattern being fairly random and nonlinear during the tests; recent NBR is faster to adapt to changing velocities.

As only ten simulations were run for each algorithm, more tests are needed to attain conclusive results. This is backed up by the fact that the standard error of the oldest neighbor runs is well within range of the error range of the recent neighbor runs. These experiments only judged the performance of the two extreme cases- oldest neighbor and most recent neighbor. It is possible that choosing a neighbor that lies in between the extremes or averaging the delta vectors of multiple neighbors could yield better performance, but this remains to be seen in future experiments.

Judging the overall performance of the tracking-velocity algorithm, it appears that, although the hit rate was always greater than the miss rate, the miss rate was significant. I attribute this high miss rate to the fact that the algorithm does not take into account the case when the predicted velocity is somewhat correct, but doesn't directly land the blimp directly over a device. Thus, I believe many of the misses are actually somewhat-correct and the actual performance of the algorithm is better than it appears.

5.2 Effect of Modifying Sensor Aperture on Tracking-velocity Performance

Although the proportion of hits to misses is virtually identical to the oldest neighbor tests with the aperture set to 0.43, the standard error is noticeably smaller. I attribute this smaller error to more consistent results created by the wider field of sensor vision. With the wider aperture, it is more difficult for the blimp to periodically slip in between sensor readings, which would cause more erratic results. The difference in standard error is an accurate illustration of the connection between physical hardware emulation in Webots and performance of the group-based tracking algorithms.

5.3 Flock Tracking Results

The flocking simulation in Webots illustrates the functionality of the move command as well as communication over multiple neighborhoods of devices. Additionally, it makes a reasonable and direct application of the velocity calculations used earlier. In terms of device hardware, communication devices must be equipped to handle large neighborhoods, as the flock rapidly converges and the number of fields being transmitted increases quickly. Simplifying flocking algorithms by reducing the number of fields would help the amount of communicated data stay reasonable. The tracking simulations used were meant only to provide an illustration of the basic Pymorphous functionalities within Webots. In terms of the actual gesture recognition and occupant recognition, more sophisticated algorithms will likely need to be used, such those provided in [4].

6. Summary

Amorphous computing allows uniform devices spread throughout space to accomplish complex goals. Additionally, the devices only use local information and scope. Using local scope and uniform devices provides a system that is very robust and not vulnerable to failure of individual devices.

Pymorphous is a Python Library with which groups of Amorphous Devices can be programmed. The Webots-runtime allows Pymorphous to use Webots essentially as if it were its own simulator. Through Webots, Pymorphous algorithms can be tested in a much less abstract environment, where device hardware can be precisely specified and its performance instantly observed. By using a less abstract simulation, countless hours of testing and dollars spent investigating performances of hardware configurations can be saved.

The functionality of the Webots-runtime was demonstrated through simple neighbor calculations, blimp velocity prediction, and mobile robot flocking. Some further areas of study could include: additional performance evaluations of Pymorphous algorithms within Webots, as well as exploration of possible overhead contributed by Webots or the Webots-runtime itself. Additionally, detailed study of communication behavior in large neighborhoods of devices with modest hardware could better prepare actual devices for possible communication lapses.

Bibliography

[1] Abelson, Hal, Jacob Beal, and Jay Sussman. *Amorphous Computing*. Thesis. Massachusetts Institute of Technology, 2007. Print.

[2] "Atan2 - C++ Reference." *Cplusplus.com - The C++ Resources Network*. Web. 05 May 2011. <http://www.cplusplus.com/reference/clibrary/cmath/atan2/>.

[3] Beal, Jacob, and Jonathan Bachrach. "MIT Proto." MIT Computer Science and Artificial Intelligence Laboratory | CSAIL. Web. 01 May 2011. <http://groups.csail.mit.edu/stpg/proto.html>.

[4] Bose, Biswajit, Xiaogang Wang, and Eric Grimson. "Multi-class Object Tracking Algorithm That Handles Fragmentation and Grouping." Thesis. Massachusetts Institute of Technology. Print.

[5] Butera, William J. *Programming a Paintable Computer*. Thesis. Massachusetts Institute of Technology, School of Architecture and Planning, Program in Media Arts and Sciences, 2002. Print.

[6] Dietrich, Charles. "Pymorphous- Python Language Extensions for Spatial Computing." Web. 01 May 2011. <http://code.google.com/p/pymorphous/>.